Science of Computer Programming 96 (2014) 177–190

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Left recursion in Parsing Expression Grammars

Sérgio Medeiros [a], Fabio Mascarenhas [b,*], Roberto Ierusalimschy [c]

[a] School of Science and Technology, UFRN, Natal, Brazil
[b] Department of Computer Science, UFRJ, Rio de Janeiro, Brazil
[c] Department of Computer Science, PUC-Rio, Rio de Janeiro, Brazil

HIGHLIGHTS

- We present a semantics for left-recursive Parsing Expression Grammars.
- A small extension adds precedence/associativity declarations to operator grammars.
- We give a semantics for compilation of left-recursive PEGs to a parsing machine.
- Our semantics are conservative: non-left-recursive PEGs are unaffected.

ARTICLE INFO

Article history:
Received 31 March 2013
Received in revised form 31 December 2013
Accepted 17 January 2014
Available online 30 January 2014

Keywords:
Parsing Expression Grammars
Parsing
Left recursion
Parsing machine
Packrat parsing

ABSTRACT

Parsing Expression Grammars (PEGs) are a formalism that can describe all deterministic context-free languages through a set of rules that specify a top-down parser for some language. PEGs are easy to use, and there are efficient implementations of PEG libraries in several programming languages.

A frequently missed feature of PEGs is left recursion, which is commonly used in Context-Free Grammars (CFGs) to encode left-associative operations. We present a simple conservative extension to the semantics of PEGs that gives useful meaning to direct and indirect left-recursive rules, and show that our extensions make it easy to express left-recursive idioms from CFGs in PEGs, with similar results. We prove the conservativeness of these extensions, and also prove that they work with any left-recursive PEG.

PEGs can also be compiled to programs in a low-level *parsing machine*. We present an extension to the semantics of the operations of this parsing machine that let it interpret left-recursive PEGs, and prove that this extension is correct with regard to our semantics for left-recursive PEGs.

1. Introduction

Parsing Expression Grammars (PEGs) [1] are a formalism for describing a language's syntax, based on the TDPL and GTDPL formalisms for top-down parsing with backtracking [2,3], and an alternative to the commonly used Context Free Grammars (CFGs). Unlike CFGs, PEGs are unambiguous by construction, and their standard semantics is based on recognizing instead of deriving strings. Furthermore, a PEG can be considered both the specification of a language and the specification of a top-down parser for that language.

PEGs use the notion of *limited backtracking*: the parser, when faced with several alternatives, tries them in a deterministic order (left to right), discarding remaining alternatives after one of them succeeds. This *ordered choice* characteristic makes PEGs unambiguous by design, at the cost of making the language of a PEG harder to reason about than the language of

* Corresponding author.
E-mail addresses: sergio@ufs.br (S. Medeiros), mascarenhas@ufrj.br (F. Mascarenhas), roberto@inf.puc-rio.br (R. Ierusalimschy).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2014.01.013

a CFG. The mindset of the author of a language specification that wants to use PEGs should be closer to the mindset of the programmer of a hand-written parser than the mindset of a grammar writer.

In comparison to CFGs, PEGs add the restriction that the order of alternatives in a production matters (unlike the alternatives in the EBNF notation of CFGs, which can be reordered at will), but introduce a richer syntax, based on the syntax of extended regular expressions. Extended regular expressions are a version of regular expressions popularized by pattern matching tools such as `grep` and languages such as Perl. PEGs also add *syntactic predicates*, a form of unrestricted lookahead where the parser checks whether the rest of the input matches a parsing expression without consuming the input.

As discussed by Ford [4], with the help of syntactic predicates it is easy to describe a scannerless PEG parser for a programming language with reserved words and identifiers, a task that is not easy to accomplish with CFGs. In a parser based on the CFG below, where we omitted the definition of *Letter*, the non-terminal *Id* could also match `for`, a reserved word, because it is a valid identifier:

$$\begin{aligned} \mathit{Id} &\rightarrow \mathit{Letter}\ \mathit{IdAux} \\ \mathit{IdAux} &\rightarrow \mathit{Letter}\ \mathit{IdAux} \mid \varepsilon \\ \mathit{ResWord} &\rightarrow \mathit{if} \mid \mathit{for} \mid \mathit{while} \end{aligned}$$

As CFGs do not have syntactic predicates, and context-free languages are not closed under complement, a separate lexer is necessary to distinguish reserved words from identifiers. In case of PEGs, we can just use the not predicate !, as below:

$$\begin{aligned} \mathit{Id} &\leftarrow !(\mathit{ResWord})\ \mathit{Letter}\ \mathit{IdAux} \\ \mathit{IdAux} &\leftarrow \mathit{Letter}\ \mathit{IdAux} \,/\, \varepsilon \\ \mathit{ResWord} &\leftarrow \mathit{if} \,/\, \mathit{for} \,/\, \mathit{while} \end{aligned}$$

Moreover, syntactic predicates can also be used to better decide which alternative of a choice should match, and address the limitations of ordered choice. For example, we can use syntactic predicates to rewrite context-free grammars belonging to some classes of grammars that have top-down predictive parsers as PEGs without changing the gross structure of the grammar and the resulting parse trees [5].

The top-down parsing approach of PEGs means that they cannot handle left recursion in grammar rules, as they would make the parser loop forever. Left recursion can be detected structurally, so PEGs with left-recursive rules can be simply rejected by PEG implementations instead of leading to parsers that do not terminate, but the lack of support for left recursion is a restriction on the expressiveness of PEGs. The use of left recursion is a common idiom for expressing language constructs in a grammar, and is present in published grammars for programming languages; the use of left recursion can make rewriting an existing grammar as a PEG a difficult task [6]. While left recursion can be eliminated from the grammar, the necessary transformations can extensively change the shape of the resulting parse trees, making posterior processing of these trees harder than it could be.

There are proposals for adding support for left recursion to PEGs, but they either assume a particular PEG implementation approach, *packrat parsing* [7], or support just direct left recursion [8]. Packrat parsing [9] is an optimization of PEGs that uses memoization to guarantee linear time behavior in the presence of backtracking and syntactic predicates, but can be slower in practice [10,11]. Packrat parsing is a common implementation approach for PEGs, but there are others [12]. Indirect left recursion is present in real grammars, and is difficult to rewrite the grammar to eliminate it, because this can involve the rewriting of many rules [6].

In this paper, we present a novel operational semantics for PEGs that gives a well-defined and useful meaning for PEGs with left-recursive rules. The semantics is given as a conservative extension of the existing semantics, so PEGs that do not have left-recursive rules continue having the same meaning as they had. It is also implementation agnostic, and should be easily implementable on packrat implementations, plain recursive descent implementations, and implementations based on a parsing machine.

We also introduce *parse strings* as a possible semantic value resulting from a PEG parsing some input, in parallel to the parse trees of context-free grammars. We show that the parse strings that left-recursive PEGs yield for the common left-recursive grammar idioms are similar to the parse trees we get from bottom-up parsers and left-recursive CFGs, so the use of left-recursive rules in PEGs with our semantics should be intuitive for grammar writers.

A simple addition to our semantics lets us describe expression grammars with multiple levels of operator precedence and associativity more easily, in the style that users of LR parser generators are used to.

Finally, PEGs can also be compiled to programs in a low-level *parsing machine* [12]. We present an extension to the semantics of the operations of this parsing machine that let it interpret left-recursive PEGs, and prove that this extension is correct with regards to our semantics for left-recursive PEGs.

The rest of this paper is organized as follows: Section 2 presents a brief introduction to PEGs and discusses the problem of left recursion in PEGs; Section 3 presents our semantic extensions for PEGs with left-recursive rules; Section 4 presents a simple addition that makes it easier to specify operator precedence and associativity in expression grammars; Section 5 presents left recursion as an extension to a parsing machine for PEGs; Section 6 reviews some related work on PEGs and left recursion in more detail; finally, Section 7 presents our concluding remarks.